foxPSL: A Fast, Optimized and eXtended PSL implementation

Magliacane, S.; Stutz, P.; Groth, P.; Bernstein, A.

published in
International Journal of Approximate Reasoning
2015

DOI (link to publisher)
10.1016/j.ijar.2015.05.012

document version
Publisher's PDF, also known as Version of record

International Journal of Approximate Reasoning 67 (2015) 111–121

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar

foxPSL: A Fast, Optimized and eXtended PSL implementation

Sara Magliacane [a,*], Philip Stutz [b], Paul Groth [c], Abraham Bernstein [b]

[a] VU University Amsterdam, Netherlands
[b] University of Zurich, Switzerland
[c] Elsevier Labs, Netherlands

ARTICLE INFO

Article history:
Received 16 December 2014
Received in revised form 15 April 2015
Accepted 28 May 2015
Available online 5 June 2015

Keywords:
PSL
ADMM
Large-scale graph processing

ABSTRACT

In this paper, we describe *foxPSL*, a fast, optimized and extended implementation of Probabilistic Soft Logic (PSL) based on the distributed graph processing framework SIGNAL/COLLECT. PSL is one of the leading formalisms of statistical relational learning, a recently developed field of machine learning that aims at representing both uncertainty and rich relational structures, usually by combining logical representations with probabilistic graphical models. PSL can be seen as both a probabilistic logic and a template language for hinge-loss Markov Random Fields, a type of continuous Markov Random fields (MRF) in which Maximum a Posteriori inference is very efficient, since it can be formulated as a constrained convex minimization problem, as opposed to a discrete optimization problem for standard MRFs. From the logical perspective, a key feature of PSL is the capability to represent soft truth values, allowing the expression of complex domain knowledge, like degrees of truth, in parallel with uncertainty.

foxPSL supports the full PSL pipeline from problem definition to a distributed solver that implements the Alternating Direction Method of Multipliers (ADMM) consensus optimization. It provides a Domain Specific Language that extends standard PSL with a class system and existential quantifiers, allowing for efficient grounding. Moreover, it implements a series of configurable optimizations, like optimized grounding of constraints and lazy inference, that improve grounding and inference time.

We perform an extensive evaluation, comparing the performance of *foxPSL* to a state-of-the-art implementation of ADMM consensus optimization in GraphLab, and show an improvement in both inference time and solution quality. Moreover, we evaluate the impact of the optimizations on the execution time and discuss the trade-offs related to each optimization.

1. Introduction

Probabilistic Soft Logic (PSL) [1–4] is one of the leading formalisms of statistical relational learning, a recently developed field of machine learning that aims at representing both uncertainty and rich relational structures, usually by combining logical representations with probabilistic graphical models. Statistical relational learning has been successfully applied in collective classification, link prediction and property prediction, in a variety of domains from social network analysis to knowledge base construction [5].

* Corresponding author.
E-mail address: s.magliacane@vu.nl (S. Magliacane).

http://dx.doi.org/10.1016/j.ijar.2015.05.012

PSL can be seen as both a probabilistic logic and a template language for hinge-loss Markov random fields, a type of continuous Markov random fields with hinge-loss potentials. Similarly to other statistical relational learning formalisms such as Markov Logic Networks (MLN), the first order logic formulae representing the templates can be instantiated (grounded) using the individuals in the domain, creating a Markov random field on which we can perform inference tasks. A key feature of PSL is the capability to represent and combine soft truth values, i.e. truth values in the interval $[0, 1]$, allowing the expression of degrees of truth within complex domain knowledge, in parallel with the uncertainty represented by the probabilistic rules. Given the continuous nature of the truth values, the use of Lukasiewicz operators, and the restriction of logical formulae to Horn clauses with disjunctive heads, Maximum a Posteriori (MAP) inference in PSL can be formulated as a constrained convex minimization problem. This problem can be cast as a consensus optimization problem [3,4] and solved efficiently with distributed algorithms such as the Alternating Direction Method of Multipliers (ADMM), recently popularized by [6].

The current reference implementation of PSL, described in [3,4], is limited to running on one machine, which constrains the solvable problem sizes. In [7], PSL is used as a motivation for the development of ACO, a vertex programming algorithm for ADMM consensus optimization that works in a distributed environment. In that work, GraphLab [8] is used as the underlying processing framework.

In this paper we extend the work initially presented in a workshop paper [9]. Specifically, we introduce *foxPSL*,[1] to the best of our knowledge the first end-to-end distributed implementation of PSL that provides an environment for working with large PSL domain, and demonstrate an improvement on the ACO system. Like [7], we adopted a distributed graph processing framework for the basis of our implementation. Instead of GraphLab, we implemented ADMM consensus optimization in SIGNAL/COLLECT [10]. Furthermore, we provided a domain specific language (DSL) that extends PSL with a class system, partially grounded rules and existential quantification.

On top of the contributions sketched in the workshop paper, we introduce a series of optimizations to *foxPSL* and improve the preliminary evaluation presented in [9] by an extensive evaluation, which includes exploring two use cases and investigating the impact of the newly introduced optimizations. In the following, we describe *foxPSL* and its features, present an empirical evaluation, and conclude with a discussion of future work.

2. The foxPSL language

As input to *foxPSL* a user can describe the problem in terms of the *foxPSL* language, an intuitive DSL for extended PSL. A problem description consists of a definition of individuals, predicates, facts and rules. In the following, we comment some lines of an example that one can find in the repository.[2]

2.1. Individuals and classes

In *foxPSL* we can explicitly list individuals in the domain, and optionally assign them to a class. For example:

```
class Person:       anna, bob
class Party:        demo, repub
class LivingBeing:  kitty, anna, bob
class Course:       ai, db
individuals:        ufo
```

By convention individuals always start with a lower-case letter. This distinguishes them from variables, which always start with an upper-case letter. Our domain consists of eight individuals: two individuals of class Person (*anna*, *bob*), two of class Party (*demo*, *repub*), three of class LivingBeing (*anna*, *bob*, *kitty*), two of class Course (*ai*, *db*) and one individual without any class (*ufo*). Classes are not mutually exclusive and the same individual can have multiple classes (*anna* and *bob* are both members of LivingBeing and Person). In addition to explicit class assignment, *foxPSL* automatically infers individuals and their classes from facts.

2.2. Predicates and predicate properties

For each predicate, we can optionally specify the classes of its arguments:

```
predicate: professor(_)
predicate: teaches(Person, Course, Person)
```

In the example, the predicate *professor* takes an argument of any class, while *teaches* takes a first argument of class Person, a second of class Course and a third of class Person. As we will see in Section 3, this represents a useful hint for the grounding phase, in which first order formulae are grounded (instantiated) with all the possible individuals in the domain.

[1] Apache 2.0 licensed, https://github.com/uzh/fox.
[2] https://github.com/uzh/fox/blob/master/examples/feature-complete.psl.

Using the class information, the only individuals that will be used to ground *teaches* will be of class Person for the first and third argument (i.e. *anna*, *bob*) and of class Course for the second argument (i.e. *ai*, *db*), greatly reducing the number of grounded predicates produced with respect to a class-less predicate.

As in standard PSL, we can define predicate properties such as functionality, partial functionality and symmetry. These properties are translated into constraints on grounded predicates during the grounding.

```
predicate [Functional]: votes(Person, Party)
predicate [Symmetric] : friends(Person, Person)
```

The functional property of *votes* means that the votes for different parties that a certain person can cast must sum up to 1. The symmetry of *married* means that for all individuals a, b, if $married(a, b) = x$ then $married(b, a) = x$.

Additionally, we can define a prior for each predicate, representing the starting truth value of a grounded predicate.

```
predicate [prior = 0.5]: young(LivingBeing)
predicate [prior = 0.0]: retired(Person)
```

In the example, a certain grounded predicate of predicate *young*, e.g. *young*(*kitty*), will be assigned a truth value of 0.5 if there is no evidence in the knowledge base for another value. A grounded predicate of *retired* will be assigned a starting truth value of 0.0, which is also the default in *foxPSL*.

2.3. Facts and implicitly defined individuals

Once we have defined the predicates, we can state some facts about our domain and their truth values. If the truth value is not mentioned, we consider it to be 1.

```
fact:                    professor(bob)
fact [truthValue = 0.8]: friends(anna, bob)
fact [0.9]:              !votes(anna, greenparty)
```

In our domain, *bob* is a professor with truth value 1 and *anna* is a friend of *bob* with truth value 0.8. Since *friends* is a symmetric predicate, it means that we expect *friends*(*bob*, *anna*) to be true with the same truth value. Moreover, *anna* does not (negation !) vote for *greenparty* with truth value 0.9 (we can omit *truthValue*), which means $votes(anna, greenparty) = 0.1$. Although *greenparty* was not mentioned as a Party before, it will be inferred as such because it is the second argument in a *votes* fact.

2.4. Rules, existential quantifiers and partial groundings

The core of *foxPSL* is the definition of rules, which are Horn rules with disjunctive heads. The restriction to this form is a constraint of standard PSL that enables the translation to convex functions. In the following example, the upper-case names represent the variables in the rules.

```
rule [weight=5]: votes(A,P) & friends(A,B) => votes(B,P)
rule [3]: young(L) => !retired(L)
```

Similarly to standard PSL, each rule can have an associated weight that represents how much it would cost to be violated. If the weight is not specified, we consider the rule a hard rule, therefore a rule that must be always true in the domain. As an implementation choice, we ground each variable in a rule only to individuals that are part of all of the classes of the predicate arguments in which the variable appears. For example, in the second rule L is only grounded with individuals in the intersection of the classes LivingBeing and Person, because it appears in both *young* which requires a LivingBeing and *retired* which requires a Person.

We allow rules that are partially grounded, i.e. that contain both individuals and variables:

```
rule [weight=3]: member(A, wwf) => votes(A, greenparty)
```

Here *wwf* and *greenparty* are both individuals (since they start with a lower case letter), while A is a normal variable.

In addition to standard PSL, we introduce the existential quantifier *EXISTS*[*variables*], which can only appear in the head, in order to preserve convexity. Given the finite domain and class system, we can safely substitute the existential quantifiers by unrolling them to disjunctions.

```
rule: professor(P) => EXISTS [C,S] teaches(P,C,S) | retired(P)
```

Similarly to other variables in the rule, the existential quantifier is grounded using the intersection of the classes of the predicate arguments where the variable appears. For example C is grounded to any individual of class Course, while S is grounded to any individual of class Person. In our current domain, the above rule can be safely rewritten as:

```
rule: professor(P) => teaches(P,ai,bob) | teaches(P,ai,anna) |
 teaches(P,db,bob) | teaches(P,db,anna) | retired(P)
```

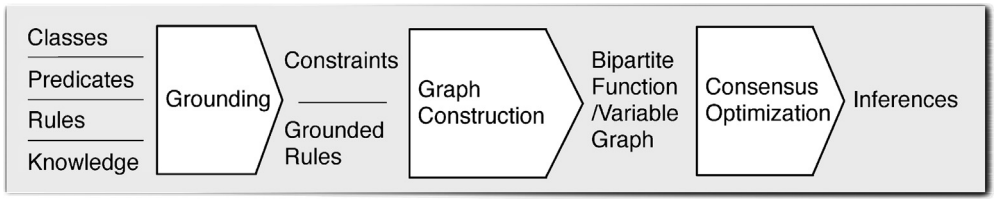

Fig. 1. The architecture of the *foxPSL* system is a pipeline that takes the rules, instance knowledge and metadata as inputs and through a series of processing steps computes the inferences.

3. System description

foxPSL is designed as a pipeline, consisting of a grounding, graph construction and inference/consensus optimization phase, similarly to most PSL implementations. The grounding phase is centralized, while the graph construction and inference are implemented in the SIGNAL/COLLECT graph processing framework. In this section we describe the underlying graph processing framework and each stage of the pipeline (see Fig. 1) with its inputs and outputs.

3.1. Graph processing in Signal/Collect

SIGNAL/COLLECT [10][3] is a parallel and distributed graph processing framework. Akin to GraphLab [8], it allows the formulation of computations in terms of vertex centric methods. In SIGNAL/COLLECT functions are separated into ones for aggregating received messages (collecting) and for computing the messages that are sent along edges (signaling). In contrast to GraphLab, SIGNAL/COLLECT supports different vertex types for different processing tasks, which allows for the natural representation of bipartite ADMM consensus optimization graphs by using two vertex types. SIGNAL/COLLECT also supports configurable convergence detection that can be based both on local and global properties. The global convergence detection is based on efficient MapReduce-style aggregations. Additional features such as dynamic graph modifications and vertex placement heuristics to reduce communication over the network are supported and might enable future *foxPSL* extensions (see Section 6).

3.2. Grounding

Grounding is well studied in a number of communities, especially in logic programming. In most probabilistic logics based on probabilistic graphical models (MLN, PSL, etc.), first order logic formulae are instantiated (grounded) using the individuals in the domain, creating a propositional representation that is used to generate a graphical model.

Given *foxPSL* class system and language extensions, the grounding procedure is different than in the mentioned work. For each predicate that is mentioned in a rule we instantiate each argument all suitable individuals, creating several grounded predicates from a single predicate. For example, for the rule:

```
rule [weight=5]: votes(A,P) & friends(A,B) => votes(B,P)
```

the predicate $votes(A, P)$ produces 6 grounded predicates: $votes(anna, demo)$, $votes(anna, repub)$, $votes(anna, greenparty)$, $votes(bob, demo)$, $votes(bob, repub)$, $votes(bob, greenparty)$.

In this phase we leverage class information of *foxPSL* to reduce the number of groundings from any individual to only the ones that match the classes of arguments. Some of these grounded predicates are user provided facts with truth values (e.g. $votes(anna, greenparty) = 0.1$), but most of them have an unknown truth value that we will compute.

Substituting the grounded predicates in all combinations in each rule, we can generate several grounded rules. The above rule generates 18 grounded rules, e.g.

$$weight = 5 : votes(anna, demo) \& friends(anna, bob) => votes(bob, demo) \quad (1)$$

The same substitution is done for all constraints. For example, the functional property of *votes* gets grounded to 2 functional constraints, e.g. for *anna*: $votes(anna, demo) + votes(anna, repub) + votes(anna, greenparty) = 1$.

3.3. Graph construction

The grounding step produces a set of grounded rules and constraints, each containing multiple instances of grounded predicates. As defined in [1,3], each grounded rule is translated to a potential of a hingeloss Markov random field (HL-MRF), a type of continuous Markov random field with constraints, using Lukasiewicz operators:

[3] http://uzh.github.io/signal-collect/.

$$x \wedge y = max(0, x + y - 1)$$

$$x \vee y = min(1, x + y)$$

$$\neg x = 1 - x$$

Using these operators on a grounded rule in the form:

$$[weight]\ b_1 \wedge \ldots \wedge b_n => h_1 \vee \ldots \vee h_m \tag{2}$$

the authors [1,3] define the distance to satisfaction as:

$$max(0, b_1 + \ldots + b_n - n + 1 - h_1 - \ldots - h_m)^p \tag{3}$$

where $p = 1$ for rules with linear distance measures and $p = 2$ for rules with squared distance measures. The intuition is that a squared distance measure is more forgiving when the violation is small and more penalizing when it is large. In *foxPSL* we can define for each rule its own distance measure at the moment of its declaration by specifying the parameter *distanceMeasure*. If nothing is specified, the default distance measure is squared. For example, the grounded rule from the above subsection, becomes:

$$max(0, votes(anna, demo) + friends(anna, bob) - 1 - votes(bob, demo))^2 \tag{4}$$

The weighted sum of the potentials defines the energy function of the HL-MRF with a domain constrained by the grounded constraints. As in standard Markov random fields, the probability is defined as an exponential of the energy function, therefore finding the assignment of unknown variables that maximizes the probability of a certain world (Maximum a Posteriori or MAP inference) is equivalent to minimizing the energy function.

Since the potentials are by construction convex functions and the constraints are defined to be linear functions, MAP inference is a constrained convex minimization problem. Following the approach described in [3,4,7] we solve the MAP inference using consensus optimization. We represent the problem as a bipartite graph, where grounded rules and constraints are represented with function (also called subproblem) vertices and grounded predicates are represented with consensus variable vertices. Each function (grounded rule or constraint) has a bidirectional edge pointing to every consensus variable (grounded predicate) it contains.

3.4. Inference: consensus optimization with ADMM

The function/consensus variable vertices implement the Alternating Direction Method of Multipliers (ADMM) consensus optimization [6]. Intuitively, in each iteration each function is minimized separately based on the consensus assignment from the last iteration and the related Lagrange multipliers. Once done, the function vertex sends a message with the preferred variable assignment to all connected consensus variables. Each consensus variable computes a new consensus assignment based on the values from all connected functions by averaging the received values, and sends it back. This process is repeated until convergence, or in the approximate case, until the primal and dual residuals fall below a threshold based on the parameters $\epsilon_{abs}, \epsilon_{rel}$. These stopping criteria are also recommended and described in [6]. Since each variable represents a grounded predicate, the assignment to a variable is the inferred soft truth value for that grounded predicate. At the end of the computation the system returns all inferred truth values.

3.5. Termination: convergence criteria comparison with ACO

As described above, *foxPSL* stops when the total primal and dual residuals are below the threshold defined in [6]. This is detected by using Signal/Collect global convergence detection, which is configured to compute the global primal and dual residuals after every iteration and compare them with the thresholds. The residual computation is implemented as a MapReduce-style aggregation over the vertices.

The implementers of ACO [7] argue that such global aggregations are wasteful, because the residual contributions of many parts of a graph often do not change much after a few iterations. For this reason they instead implement a local convergence detection algorithm in which each consensus variable computes a local approximation of the residuals using its local copies. If the local residuals in a consensus variable are smaller than a local threshold, it does not communicate to the related subproblem. If the subproblem is not awakened by any of its consensus variables, then it isn't scheduled again until there is a change in the consensus variables. The global computation stops once all local computations have converged. The disadvantage of this approach is that certain local computations may converge too eagerly, requiring others, possibly involved in more complex negotiations, to run for more iterations in order to achieve overall residuals that are below a certain global threshold. We suspect that this is the main reason for the difference in solution quality and convergence between ACO and *foxPSL* in the experiments in Section 5.

4. Optimizations

foxPSL implements a series of configurable optimizations for each phase of the system. As we discuss in Section 5, the optimizations do not improve the inference time on all settings and some may require tuning to be effective.

Algorithm 1 Lazy inference algorithm: subproblem vertex.

```
subproblem vertex, state at time t: variables x_1, ..., x_n, Lagrange multipliers y_1, ..., y_n, threshold δ
var sendSelfAwakeningMessage = false;
for i = 1 to n do
  if |x_i^t − x_i^{t−1}| ≥ δ then
    send message x_i^t to consensus variable z_i
  else
    if |y_i^t − y_i^{t−1}| ≥ δ then
      sendSelfAwakeningMessage = true;
    end if
  end if
end for
if sendSelfAwakeningMessage and no other message sent then
  send a message to self to ensure awakening
end if
```

4.1. Grounding and graph construction phases: optimizations on constraints

We implement three optimizations on the constraints: removing symmetric constraints, removing trivial functional constraints and pushing trivial partial functional constraints to bounds on the consensus variables.

Symmetric constraints are produced from predicates with the symmetric property: for example, the symmetry of *married* means that for all individuals a, b, if $married(a, b) = x$ then $married(b, a) = x$. In PSL this is implemented as a constraint in the form: $married(a, b) - married(b, a) = 0$. This optimization simply merges the two grounded predicates and uses $married(a, b)$ in all the instances where a grounded rule would contain $married(b, a)$, reducing the number of constraints and grounded predicates. Counter-intuitively, in the experimental evaluation we show that this simple optimization does not always improve the inference time, although it greatly reduces the grounding time.

Functional constraints are produced from predicates with the functional property, e.g. *votes*. In case only one of the grounded predicates in the grounded constraint, e.g. $votes(anna, demo)$, is not part of the facts and thus not yet bound, we can simply assign its truth value to $1 - votes(anna, repub) + votes(anna, greenparty)$.

Partial functional constraints are similar to functional constraints, with the difference that they require an inequality. In other words, if *votes* is partial functional, the sum of votes that a voter can cast has to be less than or equal to 1, including the possibility that the voter does not vote. In this case, we cannot assign the value of an unknown grounded predicate, apart from the trivial case in which the truth value is required to be lower than 0. In other cases we can implement the constraint as a bound on the truth value of the consensus variable. This bound will be used in each iteration by the consensus variable to clip the truth value. This optimization improves the one implemented in [7], which uses a similar mechanism to bound the truth value of any consensus variable to the interval $[0, 1]$.

An additional trivial optimization we perform is to remove grounded rules that have a distance to satisfaction of 0 for any possible assignment of the grounded predicates in $[0, 1]$, since they do not influence the minimization.

4.2. Inference phase: lazy inference and configurable steps for global convergence detection

We propose two optimizations: lazy inference and a configurable step for global convergence detection.

The lazy inference optimization is similar in spirit to the local convergence detection in [7], allowing for the computation to stop in parts of the graph while it is still active in others. The pseudo-code is shown in Algorithm 1. The main idea is that consensus variables always answer with a message, while each subproblem vertex tries to avoid resending the same message in two successive iterations. If any vertex in Signal/Collect does not receive any message, then it is not scheduled for execution. If a consensus variable is awakened by a message and does not receive any message from a certain subproblem vertex, it will compute the new consensus using an old message for that specific vertex, and send it to all the connected subproblems vertices.

Sending a message only if the message is different from the previous one creates a problem: it is possible that the internal state of the subproblem, the multipliers, are evolving, even if this is not reflected in the messages. If one does not send a message in this case, then that means that the subproblem vertex potentially does not get scheduled again, which could lead to incorrect results. On the other hand, if one sends a message to the consensus vertex whenever only the multiplier changes, then this would always trigger a potentially wasteful rescheduling of *all* subproblem vertices that are connected to that consensus vertex. To overcome this issue we propose the following algorithm: if none of the messages to the connected consensus vertices have changed, but at least one of the multipliers did change, then the subproblem vertex ensures its own rescheduling by sending a special message to itself. We implement this approach with a configurable threshold under which the change is not considered significant. This allows one to choose a trade-off between quick local convergence at the cost of slightly reduced solution quality.

In the standard ADMM algorithm and in ACO [7] convergence detection is performed after each iteration. Since global convergence detection is computationally expensive, there is a trade-off between running it every iteration, thus stopping the computation as soon as possible, and running it every n iterations, thus risking to run the computation for $n - 1$

Table 1
Dataset comparison. The Social Network datasets are from [7].

Dataset	|*subproblems*|	|*consensusVar*|	|*edges*|
Voter Network 1M	3′307′971	1′102′498	12′129′370
Voter Network 2M	6′656′775	2′204′994	24′422′102
Voter Network 3M	9′962′627	3′307′492	36′543′000
Voter Network 4M	13′349′751	4′409′988	48′989′000
Movies 100	397′465	2′823	2′310′250
Movies 150	2′455′662	6′437	14′502′334
Movies 200	7′956′132	11′740	48′731′492
Movies 250	20′331′433	18′167	121′878′352

iterations too many. In *foxPSL* we allow for the configuration of convergence detection step size, and in Section 5 we present experiments that illustrate the trade-off.

5. Evaluations

We present an extensive evaluation on synthetic datasets representing two use cases: a voter social network and a movie recommendation use case. In the following, we first describe the datasets and then present the comparison with ACO [7], showing gains in both inference time and solution quality. Secondly, we show the impact of the optimizations on the grounding time and inference time, as well as solution quality, explaining the tradeoffs in the choice of parameters for the optimizations. All evaluations are run on four machines, each with 128 GB RAM and two E5-2680 v2 at 2.80 GHz 10-core processors. All machines are connected with 40 Gbps Infiniband. *foxPSL* is developed in Scala and runs on version 1.8.0_05-b13 of the Java Runtime.

5.1. Evaluation datasets

We use two sets of synthetically generated datasets, Voter Networks and Movies, each containing four datasets of varying size, to evaluate the performance of our system.

The Voter Networks datasets are the sets of grounded rules used in [7] representing four synthetic social networks of increasing size modeling voter behavior, generated using a synthetic data generator described in [11]. In Table 1 we show dataset statistics, such as the number of subproblem and consensus vertices, as well as the number of edges connecting them. A more detailed description of the datasets can be found in [7].

Since the Voter Networks datasets contain already grounded rules, we cannot measure the grounding time and the impact of some of our DSL features and optimizations. Therefore, we developed our own synthetic dataset generator. We use a movie recommendation use case, in which we recommend a movie to a user based on the movies, actors and directors that we already know she likes, but also based on what movies, actors and directors the people in her social network and the similar users like. We formulate the problem in the *foxPSL* language, using features of *foxPSL* such as classes, priors, predicate properties and existential quantifiers. As predicates and rules we use:

```
predicate [prior = 0.5]: likes(User, Likable)
predicate [Functional]:  directedBy(Movie, Director)
predicate:               playsIn(Actor, Movie)
predicate [Symmetric]:   friends(User, User)

rule [1000]: EXISTS[MOVIE] directedBy(MOVIE, DIRECTOR)
rule [50]:   likes(USER, ACTOR) & playsIn(ACTOR, MOVIE) => likes(USER, MOVIE)
rule [1]:    likes(USER, MOVIE) & playsIn(ACTOR, MOVIE) => likes(USER,ACTOR)
rule [100]:  likes(USER, DIR) & directedBy(MOVIE, DIR) => likes(USER, MOVIE)
rule [10]:   likes(USER, MOVIE) & directedBy(MOVIE, DIR) => likes(USER, DIR)
rule [5]:    likes(A, LIKABLE) & friends(A, B) => likes(B, LIKABLE)
rule [0.1]:  likes(A, L1) & likes(A, L2) & likes(B, L1) => likes(B, L2)
```

For a given number of individuals, we populate the classes Actor (55.8%), Movie (33%), Director (11.2%) based on the proportions in which they are present in LinkedMDB.[4] All of these individuals constitute also the class Likable. Moreover, we make reasonable assumptions about the class Users, assuming that we have in general 10 times more Users than Likables. Given a proportion of known facts, we create the facts by sampling the individuals that are connected by a predicate using a normal distribution with a certain variance. We also sample the truth value of each fact using a normal distribution with mean 0.5 and standard deviation 0.25.

[4] http://wiki.linkedmdb.org/Main/Statistics.

Using the synthetic dataset generator, we produce four Movies datasets of increasing size, generated using 100, 150, 200 and 250 individuals, a mean proportion of known facts of 0.3 and a standard deviation of 0.1. In Table 1, we show the same statistics as for the Voter Network case also for the Movies datasets. The biggest datasets of each use case are comparable regarding the total number of vertices. In general the Movies datasets have a higher number of edges and a higher proportion of subproblems relative to the number of consensus variables.

5.2. Comparison with ACO

We compare *foxPSL* with ACO, the GraphLab ADMM consensus optimization implementation presented in [7], measuring both the inference time and solution quality. We do this only on the Voter Network datasets, because whilst we can load an already grounded network with ACO, it does not generate a grounding from a PSL-style language. We extended *foxPSL* with a separate loading pipeline for this serialized grounding format.

Both systems run an approximate version of the ADMM algorithm, as described in [6], with the same parameters $\rho = 1$, $\epsilon_{abs} = 10^{-5}$ and $\epsilon_{rel} = 10^{-3}$. As discussed in Section 3, ACO implements a special local convergence detection technique, while *foxPSL* employs the textbook global convergence detection algorithm. We configure both systems to take advantage of the 40 virtual cores on each machine. For *foxPSL* we configure some of the optimizations: we use the lazy optimization with a threshold of 1e–9 and a convergence detection step of 10 iterations. The grounding phase optimizations cannot be applied since the input is already grounded.

5.2.1. Inference time comparison

For each of the four datasets, we measure the inference time at a fixed number of iterations for both systems. Fig. 2 shows the results averaged over ten runs, limited to 10, 100, and 1000 iterations. Since the ACO implementation performs two extra initial iterations that are not counted in the limit, the comparison is made with an iteration limit of 12, 102, and 1002 for *foxPSL*. We stop the evaluation at 1000 iterations, because *foxPSL* converges in that interval, although ACO does not. Therefore, above the limit of 1000 iterations the running time for *foxPSL* is the same, while it continues to increase for ACO without substantial improvements in quality.

The inference time of *foxPSL* is considerably better in all considered iterations, on the two bigger datasets is more than 10 times faster, as shown by the lower computation times (y-axis) in Fig. 2. Moreover, the computation time scales linearly with increasing problem size (x-axis).

5.2.2. Solution quality comparison

We also compare *foxPSL* and ACO in terms of the solution quality on the same evaluation runs discussed above. Since PSL inference is a constrained minimization problem solved with an approximate algorithm, we consider two quality measures for solution quality: the objective function value and a measure of the violation of the constraints.

In Table 2, we show a comparison of the solution quality for *foxPSL* and ACO at iteration 1000 from the previous experiments. The four parameters we compare are the objective function value *ObjVal*, the sum of violations of the constraints $\Sigma viol$, the number of violated constraints at tolerance level 0.01 $|viol@0.01|$ and the number of violated constraints at tolerance level 0.1 $|viol@0.1|$. Moreover, we report an estimation of the optimal objective function value *OptVal* as computed using lower epsilons ($\epsilon_{abs} = 10^{-8}$, $\epsilon_{rel} = 10^{-8}$) which has a sum of constraint violations on the order of only 10^{-5}. We can see that *foxPSL*'s objective function value is closer to the estimated optimal objective function value than the ACO's for all datasets. The sum of the violations of the constraints is lower in *foxPSL* by a factor of more than 10.

Besides the lower total violation of the constraints, Table 2 also shows the number of constraints that are violated by more than a certain threshold of tolerance. In particular, in the ACO solution there are several hundred constraints that are violated even while tolerating errors of 0.1, which is sizeable considering that the variables are constrained in the interval $[0, 1]$. In general, the violations of the ACO solution are larger and less spread across the constraints than the violations found in the *foxPSL* solution, possibly due to the local convergence detection of the former, which may be too eager to converge on some subgraphs, leading to a solution with a higher approximation error.

5.3. Optimization evaluation

The goal of the optimization evaluation is twofold: on the one hand we would like to find out how the optimizations affect the performance of the grounding and inference phases of *foxPSL*. On the other hand we would like to answer the question if the impact of the optimization changes with the dataset size. For this reason we first evaluate the performance for *foxPSL* with the baseline settings on datasets of increasing size. For each optimization we then change the feature relative to the baseline configuration in order to gauge the effect on the performance.

We evaluate three optimizations: the symmetric constraint optimization, the global convergence detection step and the lazy inference threshold. The other optimizations on constraints have similar results to the symmetric constraint optimization. We perform the experiments on both use cases with the exception of the symmetric constraint optimization, which cannot be used on the Voter Networks datasets because they contain already grounded rules and they do not contain any symmetric constraint. Our baseline configuration consists of all grounding phase optimizations enabled, lazy inference enabled with a threshold of 10^{13} and the same parameters as in the ACO comparison, $\rho = 1$, $\epsilon_{abs} = 10^{-5}$ and $\epsilon_{rel} = 10^{-3}$. On

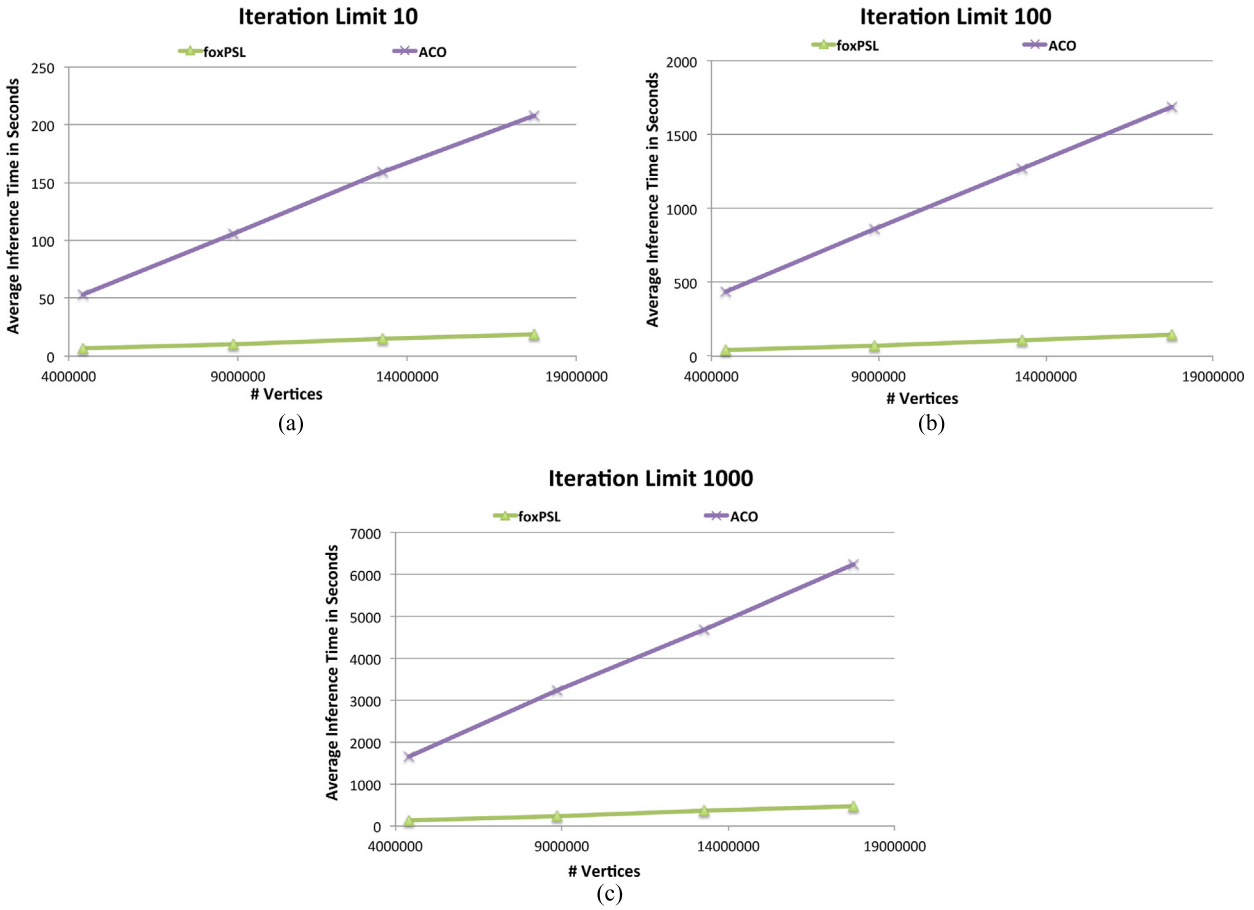

Fig. 2. (a), (b) and (c) compare the average inference time between *foxPSL* and ACO. For each graph size there were ten runs, with an iteration limit of 10, 100 and 1000 respectively.

Table 2
Comparison of the solution quality for *foxPSL* and ACO with iteration limit 1000 and same parameters ($\rho = 1$, $\epsilon_{abs} = 10^{-5}$, $\epsilon_{rel} = 10^{-3}$).

|*vertices*|	*OptVal*	*foxPSL*				ACO			
		ObjVal	Σ*viol*	|*viol*@0.01|	|*viol*@0.1|	*ObjVal*	Σ*viol*	|*viol*@0.01|	|*viol*@0.1|
4410 K	4838.14	4809.97	7.05	20	4	4722.09	119.55	491	233
8861 K	9692.03	9678.66	13.38	18	3	9520.12	233.04	924	431
13.2 M	14 521.26	14 448.14	19.77	42	6	14 199.59	349.14	1387	640
17.7 M	19 425.71	19 441.26	26.64	52	5	19 119.38	472.49	1894	863

the Voter Network dataset we use a convergence detection step of 10, while on the Movies dataset we check convergence at each step.

5.3.1. Symmetric optimization

Fig. 3 shows the impact of symmetric optimization on the Movies datasets. As already mentioned, we cannot compare on Voter Networks datasets because they contain already grounded rules and they do not contain any symmetric constraint. The optimized grounding and inference times are represented with a dashed line, while the full lines represent the grounding and inference times when the optimization is disabled. For the Movies dataset, shown in Fig. 3, the grounding time improves greatly with the optimization with gains increasing with the size of the dataset. On the other hand, the inference time is essentially the same. This is counter-intuitive, since the symmetric optimization reduces the number of vertices, therefore we would expect the inference time to improve. In practice, given the many other rules in this use case, the reduction of vertices is in the order of 0.1%, so the difference is marginal.

5.3.2. Global convergence detection step

Fig. 4 shows the impact on inference time of running the convergence detection every n iterations on both the Voter Networks and the Movies datasets. In general, there is a trade-off between saving the overhead of a too frequent con-

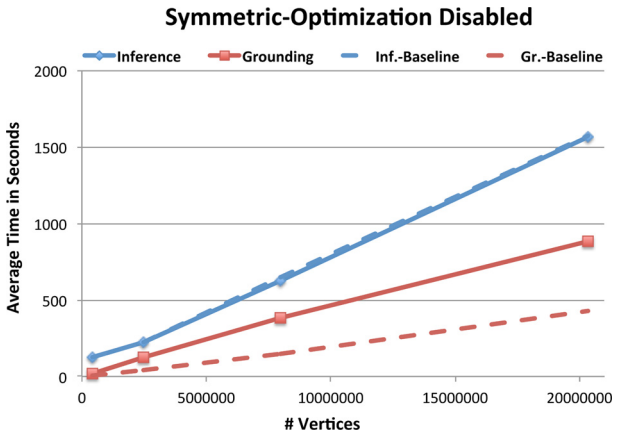

Fig. 3. Symmetric optimization evaluation on Movies datasets: dashed line represents the optimized system.

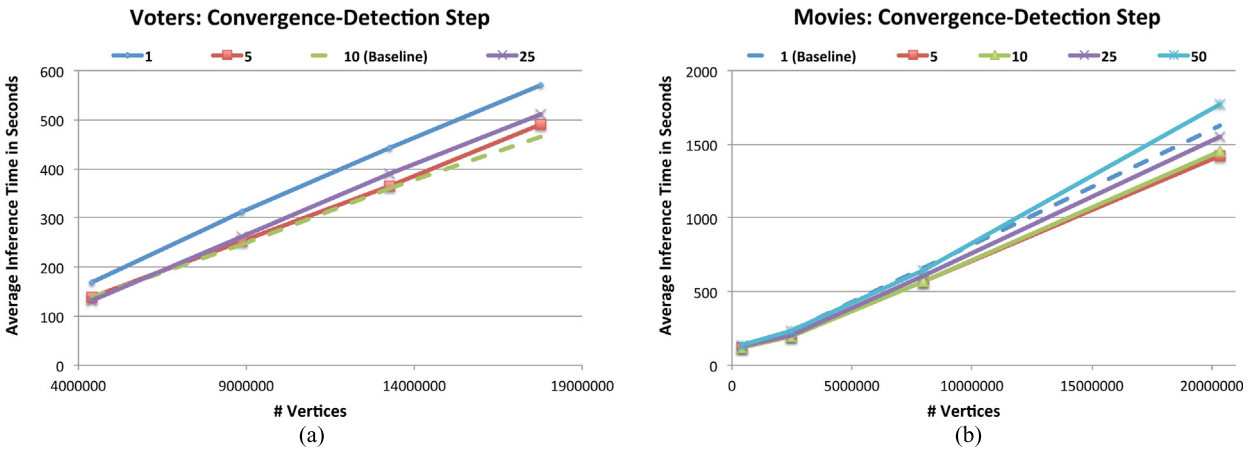

Fig. 4. Varying the convergence detection step for Voter Networks (a) and Movies (b) datasets.

vergence detection and running more iterations than necessary. In our use cases, the standard configuration in which we run the convergence detection every step is consistently outperformed the other configurations. The optimal convergence detection step seems to be between 5 and 10 iterations, decreasing the inference time by 10%, while at 25 iterations the benefits decrease. Above 25 iterations, the inference time actually worsens with respect to the baseline. We can see that the effects of this optimization increase linearly with the size of the dataset and therefore the cost of the global convergence. A positive side effect of a higher convergence step is the slight increase in solution quality, since the computation runs for few more iterations.

5.3.3. Lazy inference thresholds

Fig. 5 shows the impact of enabling lazy inference and varying the threshold under which a change is not considered significant on both the Voter Networks and the Movies datasets. On the Voter Networks datasets, shown in Fig. 5(a), enabling lazy inference reduces the inference time by 10%, while the difference between the various thresholds is not clear. In general, a higher threshold, e.g. 10^{-6}, allows for faster inference, but with a slightly lower solution quality (an objective function that is higher by 0.01 and a sum of broken constraints that is higher by 0.05 in the worst case). On the Movies datasets, shown in Fig. 5(b), the improvement is very small. We suspect that the difference is due to the homogeneity and the high number of edges of the Movies dataset, which does not allow parts of the graph to converge locally.

6. Limitations and conclusions

In this paper, we extend the work initially presented in a workshop paper [9] introducing *foxPSL*, to the best of our knowledge the first end-to-end distributed implementation of PSL that provides an environment for working with large PSL domain, including a domain specific language (DSL) that extends PSL with a class system, partially grounded rules and existential quantification. Moreover, we present a series of optimizations to *foxPSL* and improve the preliminary evaluation presented in [9] by an extensive evaluation, which includes exploring two use cases and investigating the impact of the newly introduced optimizations.

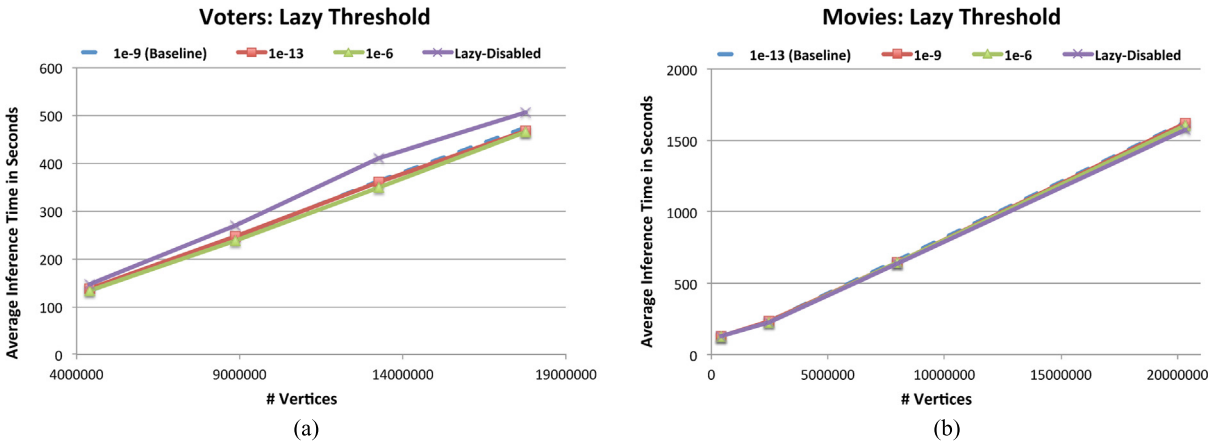

Fig. 5. Varying the lazy inference threshold for Voter Networks (a) and Movies (b) datasets.

Our current implementation has limitations. Our ADMM algorithm uses a fixed step size, leading to an initially fast approximation of the result and a slow exact convergence. An improvement would be a variant with adaptive step sizes. Additional improvements might be incremental reasoning when facts/rules change, taking advantage of the dynamic graph modification features supported by SIGNAL/COLLECT, and the use of SIGNAL/COLLECT vertex placement heuristics to reduce messaging.

Whilst developing *foxPSL*, we found that PSL provides a powerful formalism for modeling problem domains. However, its power comes with numerous interactions between its elements. Hence, the use of PSL would be aided by a DSL and an end-to-end environment that allows for the systematic analysis of these interactions. We believe that *foxPSL* is a first step towards such an environment.

Acknowledgements

We would like to thank Hui Miao for sharing the ACO source code and evaluation datasets, and Stephen Bach for being able to use his optimizer implementations in *foxPSL*, as well as for the useful discussions. Furthermore, we would like to thank the Data2Semantics project in the Dutch National Programme COMMIT and the Hasler Foundation under grant number 11072 for supporting this work.

References

[1] M. Broecheler, L. Mihalkova, L. Getoor, Probabilistic similarity logic, in: UAI, 2010.
[2] A. Kimmig, S.H. Bach, M. Broecheler, B. Huang, L. Getoor, A short introduction to probabilistic soft logic, in: NIPS Workshop on Probabilistic Programming: Foundations and Applications, 2012.
[3] S.H. Bach, M. Broecheler, L. Getoor, D.P. O'Leary, Scaling MPE inference for constrained continuous Markov random fields, in: NIPS, 2012.
[4] S.H. Bach, B. Huang, B. London, L. Getoor, Hinge-loss Markov random fields: convex inference for structured prediction, in: UAI, 2013.
[5] L. Getoor, B. Taskar, Introduction to Statistical Relational Learning, MIT Press, 2007.
[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, Distributed optimization and statistical learning via the alternating direction method of multipliers, Found. Trends Mach. Learn. 3 (1) (2011).
[7] H. Miao, X. Liu, B. Huang, L. Getoor, A hypergraph-partitioned vertex programming approach for large-scale consensus optimization, in: 2013 IEEE International Conference on Big Data, 2013.
[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, GraphLab: a new parallel framework for machine learning, in: UAI, 2010.
[9] S. Magliacane, P. Stutz, P. Groth, A. Bernstein, foxPSL: an extended and scalable PSL implementation, in: AAAI Spring Symposium on KRR, Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches, 2015.
[10] P. Stutz, A. Bernstein, W.W. Cohen, Signal/Collect: graph algorithms for the (semantic) Web, in: ISWC, 2010.
[11] M. Broecheler, P. Shakarian, V. Subrahmanian, A scalable framework for modeling competitive diffusion in social networks, in: International Conference on Social Computing (SocialCom), 2010.